DATA MANAGEMENT IN A DISTRIBUTED DESIGN MODELING ENVIRONMENT

by

JOHNNY T. CHANG

B.S. Mechanical Engineering
Massachusetts Institute of Technology, 1997

Submitted to the Department of Mechanical Engineering
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 2001

Signature of Author..... .....
Department of Mechanical Engineering
May, 2001

Certified by.....
David Wallace
Esther and Harold E. Edgerton Associate Professor of Mechanical Engineering
Thesis Supervisor

BARKER

Accepted by.....
Ain A. Sonin
Chairman, Department Committee on Graduate Students

DATA MANAGEMENT IN A DISTRIBUTED DESIGN MODELING ENVIRONMENT

by
JOHNNY T. CHANG

Submitted to the Department of Mechanical Engineering
On May 11, 2001 in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

ABSTRACT

This thesis discusses the implementation and future development of data management tools for a research software system called the Distributed Object-based Modeling Environment. The management of data in product development has evolved from simple data storage to the maintenance of complex data relationships. By providing data with descriptive attributes (metadata) and by describing key relationships between data, product data management (PDM) systems facilitate the sharing of data within development environments. However, the inherent information content and value of data is not shared.

DOME provides a framework by which the intelligence of data can be shared in a distributed network as published services. Individual data files from different applications can be encapsulated with defined interfaces, shared in a marketplace of data objects, and interconnected in design simulation network models. However, DOME presently has no PDM capabilities; this thesis presents a method of tracking DOME models, and proposes further PDM improvements to DOME. By enabling corporate data management capabilities in a tool such as DOME, ongoing development simulations and analyses can become a more integrated part of the engineering data management process. In addition, the added descriptive capabilities can make the sharing of this ongoing work more efficient through more productive searches, and therefore enrich communication within development environments.

Thesis Supervisor: David Wallace

Title: Esther and Harold E. Edgerton Associate Professor of Mechanical Engineering

ACKNOWLEDGEMENTS

Thank you for everything, Dave. I promise that I will do my best to make all this worthwhile, and make a real difference in people's lives, as I so want to do. That and free medical care for you for the rest of your life (if you trust me with a knife).

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Problem Statement

With the leaps in information technology and computing power over the last quarter century, there has been a scramble among businesses to utilize these new capabilities. In the beginning, computers were seen basically as a new way to do old things; corporate processes remained the same. Rather than using typewriters to produce documentation, one could use word processors. Rather than drafting engineering drawings by hand, one could use drafting software. Rather than storing documents in physical file folders, one could store them in electronic file folders on a computer. In recent years, however, many have realized that the key to harnessing information technology to increase productivity is in recognizing how it can transform processes. Williams notes that "the future of business is not investing in more technology, but embracing new paradigms." [Williams, 1994]

In the late 1980s, companies began to realize that simply warehousing corporate data electronically would not be sufficient, in light of the sea of data that they generated [Williams]. Far from increasing productivity, electronic data warehousing could severely hamper productivity. Information technology had not solved the problem of the data deluge, and in fact had complicated the matter by rendering the data less accessible, hidden inside some computer somewhere. However, the advent of the Oracle's relational database management system around the same time would facilitate the development of the first product data management systems to help manage the increasing complexity of electronic data storage [Greenwald, 1999].

A product data management system at the simplest level is a software application layer that allows a company to truly *manage* its data, rather than simply warehouse it. By maintaining data attributes and data relationships in addition to the data itself, a PDM system provides a view of corporate data as whole, connected, and dynamic, rather than a disjointed set of stored objects. At a higher level, a PDM system can provide features for controlling the flow of data throughout a product's life cycle; in this way, information

technology begins to scratch the surface of a new paradigm. Data becomes more than simply a stored, stagnant record of what has been done, and becomes the live substance of the work that is being done.

Nevertheless, according to company sources, even as large and complex a company as Ford Motor Company still uses its SDRC Metaphase PDM system as not much more than a glorified data warehousing system. In the face of daunting up-front capital costs, a company may elect simply to set up a corporate electronic data vault, albeit with a PDM system as the interface [Williams, 1994]. In addition, even in the best of cases, PDM systems still treat data in an administrative manner – that is, each document is treated as a somewhat unintelligent object to be categorized and managed. The actual value of the data within a document is ignored. For example, a PDM system treats an information-rich SolidWorks 3D model in basically the same way as a brief departmental memo. While the two pieces of data may be described by different attributes, and be classified and used in very different ways, the PDM system extracts effectively none of the data's inherent value.

A new paradigm is presented by DOME (Distributed Object-based Modeling Environment), developed at the Computer-Aided Design Laboratory at the Massachusetts Institute of Technology. DOME goes beyond simply *managing* data and enables users to tap into the inherent value of data by allowing individuals to define detailed parametric relationships between data. It provides a framework within which individual data objects (e.g., an engineering spreadsheet or a Solidworks model) can be interconnected to simulate the interactions between conceptual parts of a complex design. These data objects, or modules, provide their capabilities to other objects in a controlled manner via defined interfaces. Entire models consisting of many interconnected modules can then be examined using DOME's analysis tools [Pahng, 1998].

The work behind this thesis involves a marriage between conventional PDM systems and the DOME framework. While DOME presents a paradigm in how data is treated and used, and provides tools that can significantly change traditional design processes, it lacks an integrated method of simply managing the data objects that it uses, as well as the

additional information generated by analysis tools. One can consider conventional PDM systems to be a way of managing relatively raw engineering data, DOME to be a way of tapping into the intelligence of that engineering data, and a DOME data management system to be a way of managing the product of that intelligence.

1.2 Overview of the thesis document

This thesis document describes the background of the related research work, the considerations involved in implementing the DOME data manager, the implementation of the data manager, and proposed future work on data management for DOME.

Chapter two provides the background of product data management, in more detail than in Section 1.1. Relational databases and basic database concepts are provided first. A comparison between the conventional "paper" design process, the product life cycle in the presence of a conventional PDM, and the potential process change afforded by the DOME framework, is then presented.

Chapter three describes DOME in greater detail by explaining its architecture, how it works in practice, and how it can fundamentally affect the design process. DOME's data management needs are then introduced.

Chapter four covers the implementation of the DOME data manager produced for this thesis. Basic considerations such as the database system, the programming language, and the scope of the implementation are discussed. The implementation is then described in detail.

Chapter five presents proposed future work to provide true PDM capabilities within DOME. Beyond providing database connectivity and version control, a DOME database manager should maintain data attributes and relationships. A database schema that would allow this is suggested, and user interactions are described.

Chapter six delves into the potential benefits of a DOME data manager, in terms of the additional tools and features that can be developed. Workflow management, distributed search capabilities, and data markup features are discussed.

Chapter seven presents conclusions made from the study of product data management methodology with respect to DOME. The feasibility of fully integrating a PDM system into the DOME framework is discussed, as well as issues to consider in doing so. The benefits of performing such a task in terms of presenting a new design paradigm are summarized.

2 BACKGROUND

2.1 Relational Databases

Traditionally, the structure of electronic databases reflected that of physical data storage; data was stored on computers much as one would store contact information in a Rolodex, or various documents in a filing cabinet. In this hierarchical approach, data templates were constructed at the database design stage, and a piece of data consisted of several pre-defined fields; for example, a contact would be stored as a set of fields such as last name, first name, and home phone number. Pieces of data would be stored in an ordered hierarchy; for example, "college buddies" and "hockey buddies" would both be organized under the "buddies" category (see Figure 1). This traditional database model left pieces of data relatively isolated from one another, provided no significant flexibility in terms of changing the data template to accommodate the addition or removal of fields, and often resulted in widespread data inconsistencies because of the lack of data relationships and the need for data replication [Abbey, 1999]. What if a "buddy" was both a college buddy and a hockey buddy? The only solution was to duplicate data in more than one location, resulting in significant synchronization errors.

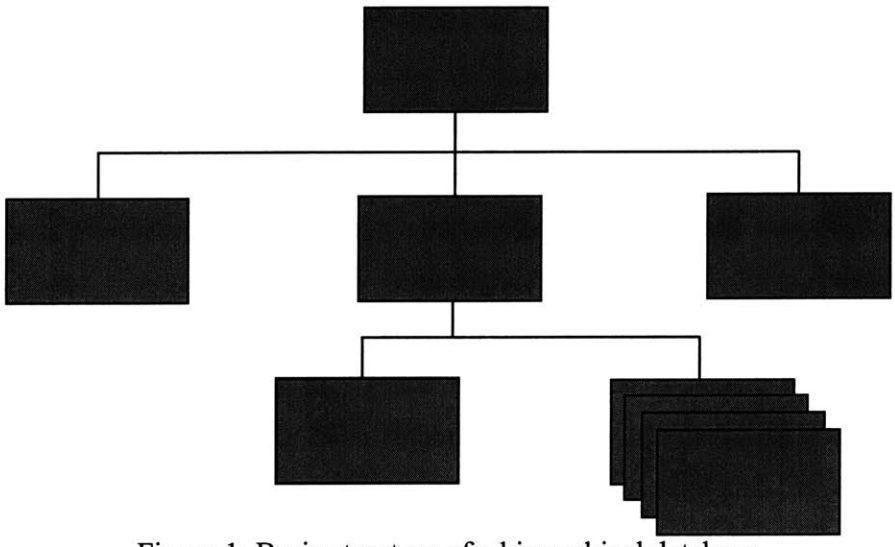

Figure 1: Basic structure of a hierarchical database

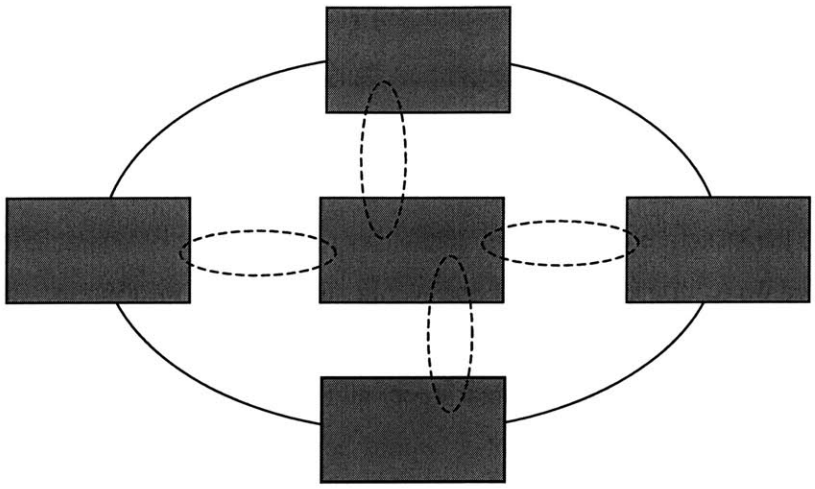

Figure 2: Basic structure of a relational database; data is maintained in a loose structure connected by data relationships

In the late 1970s, based on work previously undertaken by IBM, Oracle came into existence as the world's first relational database management system [Greenwald, 1999]. A relational database consists of sets of two-dimensional tables in which rows and columns can be linked together in complex networks. These linkages provide information about the relationships between different pieces of data (see Figure 1). For example, a "college buddies" table could reference data in a master contacts list. If a contact belongs in more than one table, each table can refer to the master list; in this way, a piece of data exists in one and only one location, and the relationships between data in different tables can be described and maintained.

Relational databases are ideally suited for data management in a complex, integrated engineering design environment. One engineering document rarely falls neatly into one spot in a hierarchy. More often than not, the same document is needed by many people in many different places at many different times. In addition, the engineering and business relationships between various documents need to be described and maintained for the warehoused data to be truly useful.

2.2 Data Management in Product Development

2.2.1 Conventional data management

In conventional product development environments, data is managed largely without the benefits of information technology. Product data and documentation are simply warehoused, and data objects may be embodied in many different forms and formats – electronic or paper, text or binary, etc. Data relationships and attributes in this situation are effectively stored in the heads of engineers and managers. The problems that arise from this rather haphazard mode of data management, or lack thereof, becomes especially apparent in complex design systems such as those encountered in the aerospace industry [Tsao, 1993].

2.2.2 Product data management systems

In the deluge of engineering documents and data produced in the design of complex systems, it becomes obvious that a better system than the filing cabinet, electronic or otherwise, is needed. Complexity in design systems affects the speed of the development process and increases the risk of errors, inconsistencies, and productivity loss due to redundancy [Bourke]. The role of the product data management system is to help deal with this complexity by maintaining data relationships and facilitating the flow of data throughout a product's development and life cycle.

For a PDM system to succeed in this role, it is clear that stored data needs to be described by more than just a filename. In product data management terminology, data that describes data is called *metadata*. In addition to warehousing a given piece of data or document, a product data management system provides the descriptive attributes of that piece of data or documentation in the form of metadata. With those attributes, the PDM system is then able to formulate and maintain relationships between different pieces of data. Altogether, a PDM system should manage engineering data, attributes of that data, as well as relationships between data and metadata [Hewlett-Packard].

Additionally, PDM systems can provide engineering *procedure* management by directing official data releases and other official actions during the development cycle. Beyond procedure management, some PDM systems provide *process* management tools, whereby the systems can have real impact on the fundamental way that the design process itself is executed, and on the workflow continuously throughout the product development cycle. By facilitating the distribution of in-progress data and allowing the controlled exchange and markup of such data, a PDM system can alter design process mentality and control the progress of the development cycle. In this case, data flow from one person to the next, or one phase to the next, is regulated by defined rules [Williams]. Process management features in PDM systems make workflows data-driven, and thus taps more into the inherent value of the data that it is managing.

2.2.3 DOME

Advancements in object-oriented concepts have taken the role of data tools beyond simple data management and metadata maintenance. In the object-oriented mentality, access to the inherent intelligence of an object is controlled and provided by a creator-defined interface. The interface delineates the inputs that the object requires in order to function, the outputs that it will provide in return, and the internal details that are visible to other objects. In the case of object-oriented programming languages such as Java and C++, this means that code is encapsulated and hidden in objects, and other objects can access only the data and functionality that make up the objects' predefined public interfaces. Similarly, in the data realm, data such as an engineering spreadsheet can be hidden and encapsulated, yet its functionality can be provided through predefined interfaces. Developments such as the Common Object Request Broker Architecture (CORBA) and Object Linking and Embedding (OLE) enable the sharing of data functionality across diverse data formats and applications [Conaway, 1995].

Riding on this object-oriented wave, the Distributed Object Modeling Environment (DOME) provides a framework by which the inherent value of data is shared instead of simply packed away into storage or lost in a sea of complexity. While PDM systems maintain additional descriptive metadata on top of data itself, they largely disregard the actual information content of the data objects. With DOME, a design team could relate the actual functionality of data objects, and not just the attributes of that data. Once a network of relationships is established, simulations and design analyses could be conducted to understand and observe interactions between different parts of a design. In essence, PDM systems pack data objects away in closed, labelled boxes; DOME wraps the data objects in a way that actually facilitates access to the contents of those boxes.

3 DOME

The Distributed Object-based Modeling Environment (DOME) is a product of the Computer-Aided Design Laboratory at the Massachusetts Institute of Technology. A commercial spin-off of the research product is presently being developed by Oculus Technologies Corporation. The software provides a framework on which the concept of sharing the inherent informational value of object-oriented data can be tested and further developed. DOME follows in the prevailing internet philosophy of providing resources and services that are accessible to users with the least possible need for client-side customization. Just as web browsers make diverse data and applications available to clients on different software platforms, DOME makes it possible to provide diverse product development services to users regardless of platform, via standard Internet communication protocols. Users can access a wide range of data without abandoning available applications, or environments familiar to them [Wallace, 2000].

The power of this data sharing, however, comes not from simply the ability to access individual data documents; rather, DOME enables the connection of distributed data objects into data networks where the outputs from one data module provide the inputs to other modules. In this way, complex network simulations can be constructed rapidly and without the potential difficulties presented by differences in platforms and geographic locations.

3.1 Architecture

DOME provides its services using a two-tiered server-client architecture that employs standard TCP/IP for communication between servers and clients, as well as among servers. Users communicate through servers via Remote Method Invocation (RMI), which connects Java applet DOME clients to servers. On the server side, C++ is used in implementing the DOME core, and communicates with the Java side using Java Native Interface (JNI). Server interactions with third-party applications occurs through custom plugins that utilize the individual applications' published Application Programming Interfaces (APIs). Some custom plugins include ones for Microsoft Excel, SDRC I-DEAS, SolidWorks, and Matlab.

3.2 DOME Model-Service Structure

A DOME service represents a data object that is available for use by DOME clients. A service is published, or made accessible to external parties, when an interface is defined for the data object. For example, an excel spreadsheet that calculates a given output for a given input can be published as a service by creating an interface with the DOME add-in within Excel (see Figure 3). Once a service interface is defined, it can be accessed from a DOME client via the DOME server (see Figure 4). In practice, services would be displayed in the client as they're being added (see Figure 5).

In addition to custom third-party application services, DOME allows the addition of certain built-in services as well. These basic types include strings, booleans, containers, and relations. Essentially, on an interface level, they provide data in the same way that a custom service does.

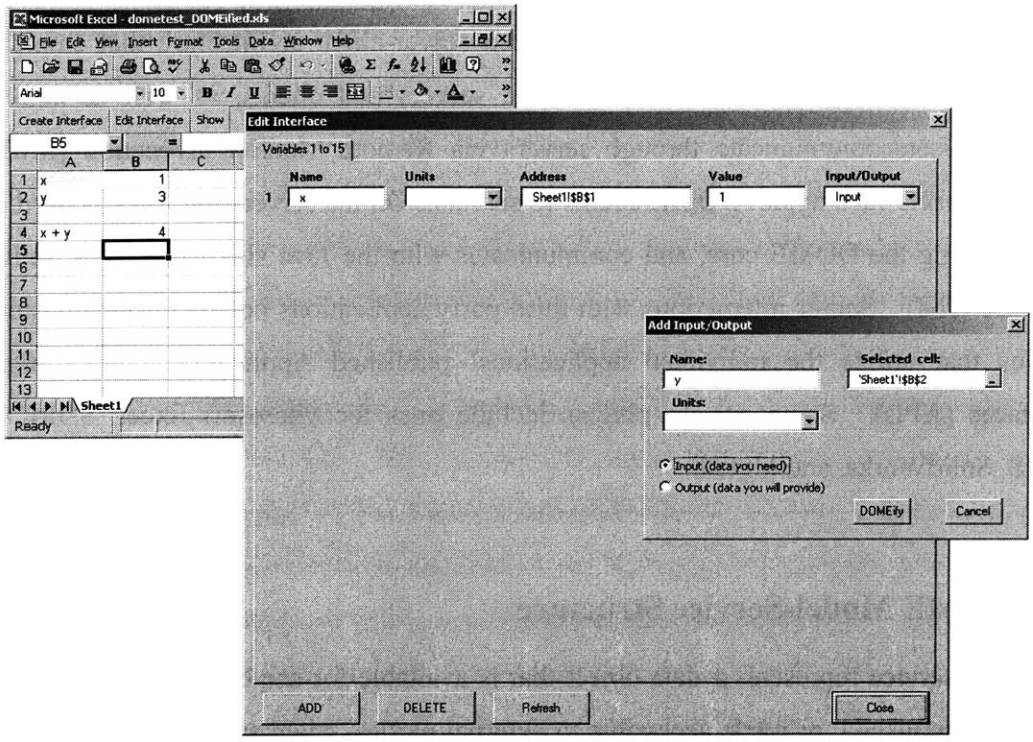

Figure 3: Creating a DOME interface for an Excel spreadsheet

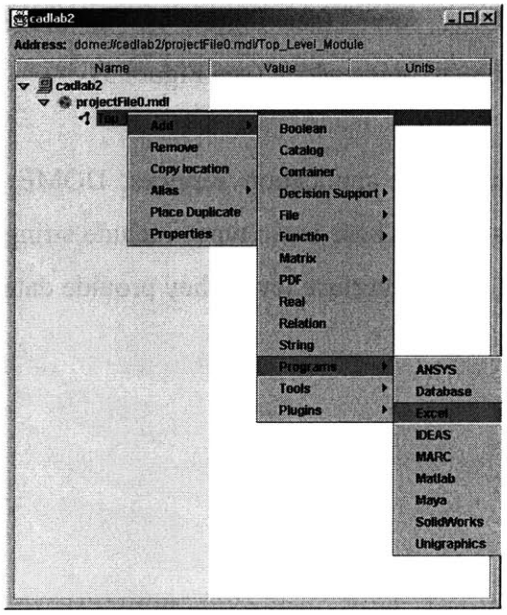

Figure 4: Adding a service to a model in the DOME client

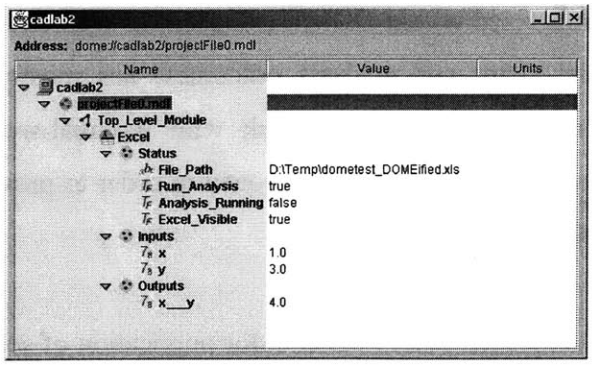

Figure 5: Excel service as displayed in DOME client

Once services have been added to the scope of a DOME client, they may be connected using relations, objects that encapsulate relationships between services ranging from simple equality statements to complex C++ code. Altogether, the services and the connections between services comprise a DOME model. These models can then be analyzed using various tools provided by DOME, and they themselves can be interconnected as well.

Throughout the process of adding services to a given model, the service user can access the full functionality and value of the data object behind the service. Yet, the details of the data object's implementation is securely encapsulated by the service interface, which allows access to only provider-defined inputs and outputs. As a result, providers can make the core value of their data objects available, yet retain intellectual property rights to the models and processes within the data object.

3.3 DOME Data Management

Management of the data produced by DOME is presently minimal at best. Research and development efforts have concentrated largely on the fundamental principle of providing distributed service sharing capabilities, and exploring the implications of those capabilities in terms of encouraging collaboration, integrated design, and the reuse of engineering services. Management of the data shared and used by DOME models

(services), the data synthesized within DOME (models), as well as the data generated by DOME (specific model states and analysis outcomes) has mostly been ignored. In addition, integration of the DOME framework with the real-world corporate PDM environment of most companies has been postponed in order to provide a simplified test environment for the DOME concept.

In the existing DOME implementation, the data for publication of services for third-party application data files are not created or stored in a consistent manner. Part of the reason for this inconsistency is application-dependent; some application APIs allow publication data to reside in the data files themselves, while others require external text files. Nonetheless, once the necessary data has been generated, DOME is aware of the services only as they are requested and located by the user. Furthermore, the only knowledge that the user can have of the service is the service interface – what inputs and outputs can be accessed and manipulated. There is no descriptive metadata for the service object, there is no way to determine relationships between available services, and there is no way of controlling the creation, modification, deletion, and use of services in the workflow.

Within DOME, data that describes synthesized models is managed in a rather crude fashion. A model is exported to a plain text file in a predefined directory on the server hard drive. This text file, called an MDL file because of its 3-letter Microsoft Windows filename extension, describes the basic components of the as service attributes organized with a series of square brackets in a hierarchical manner (see
Figure **6**). These text files constitute the extent of model data management in DOME; there is no metadata, and no versioning capability. "Opening a model" amounts to making sure that the desired MDL file is in the designated directory before the server is started, and then selecting the proper filename in the client once the server has been started.

```
MDL2 [RelationContainerModule [name Top_Level_Module]
 [PRealModule [name A] [value 0.000000]]
 [PRealModule [name B] [value 1.000000]]
 [PCompiledRelationModule [name rel] [independent B] [dependent A]
[pre #include <relationHeader.h>
#include <RelationContainerModule.h>
#include <XLRealModule.h>
#include <MATLABRealModule.h>
#include <IDEASDimensionModule.h>
#include <Catalog.h>
#include <MatrixModule.h>
#include <Segmented.h>
] [value A = B;
]]]
```

Figure 6: Sample MDL file for a DOME model

Once a model has been set up or retrieved, there is yet another layer of DOME-related data – the analytical data resulting from the model simulation. This data is the model's raison d'être in the first place. The point of providing the DOME framework and enabling data sharing and the creation of service networks is to be able to analyze the networks and obtain results and observations about particular design relationships. It would thus be beneficial to be able to record and manage those results, and particular sets of service input/output values that constitute a model state of interest. In this context, the "model" is the actual network of services and relationships, while a "model state" is the model with specific values at the service inputs and outputs.

This thesis project proposes that it is a logical progression for DOME to combine the advantages of conventional PDM systems with the DOME product development paradigm. In the case of conventional PDM systems, data is stored with descriptive metadata, but the data itself is effectively inaccessible, unless a user expressly opens one specific data file with the specific corresponding third-party application. However, PDM systems can control engineering procedures such as defined releases, as well as affect engineering workflow by enabling data to drive and shape development processes. In all cases, PDMs treat the data objects as discrete, locked boxes. On the other hand, DOME provides access to the core computational value of data objects, but does not retain descriptive metadata and data relationships, and therefore cannot yet reach its full potential to truly affect workflow, processes, and engineering procedures. The natural

evolution of the DOME framework is the incorporation of data management features with its powerful data sharing capabilities.

4 THE DOME DATA MANAGER

The ultimate goal of a full DOME data manager is to provide the following capabilities:

- integrated warehousing or tracking of all DOME data (available services, synthesized models, and resulting analysis data)
- provision of descriptive metadata for all stored data objects to improve data search and browse functionality
- maintenance of administrative metadata relationships, such as versioning, ownership, related data objects (e.g., documentation), and related services
- process and workflow control based on DOME-generated simulation and analysis results

The implementation of the DOME Data Manager developed for this thesis provides a basic framework for database connectivity, and tests the framework by providing a mechanism for the versioning of models, and for creating metadata to describe models. This chapter focuses on the basic technical issues involved in providing database connectivity, the details of the integration of the Data Manager with DOME, and a description of the Data Manager user interface and Data Manager API used to access the database. The next chapter will then explore a possible direction for a future implementation of the DOME Data Manager.

4.1 Technical Considerations

4.1.1 DOME data types

As described in section 3.3 above, the types of data objects associated with DOME can be classified into three major categories. In addition, an administrative data object category would be helpful:

- Service publication data – the third-party application files, along with accompanying publication data text files if required, that are define the service interface and are necessary for use in DOME
- DOME model structure data – data that describe the contents of a DOME model (services, relations, and containers) and how they are interconnected in a DOME simulation network
- Output analysis and model state data – data produced by the application of analysis tools to a DOME model, as well as data describing the values of independent service input/output variables within a model at a particular point in time
- Administrative data – data such as DOME users and available hosts that are not a value-added part of the product development data, but are necessary for the implementation of proper data management

Figure 7 describes these data types graphically. This implementation of the DOME Data Manager deals with the management of DOME model structure data.

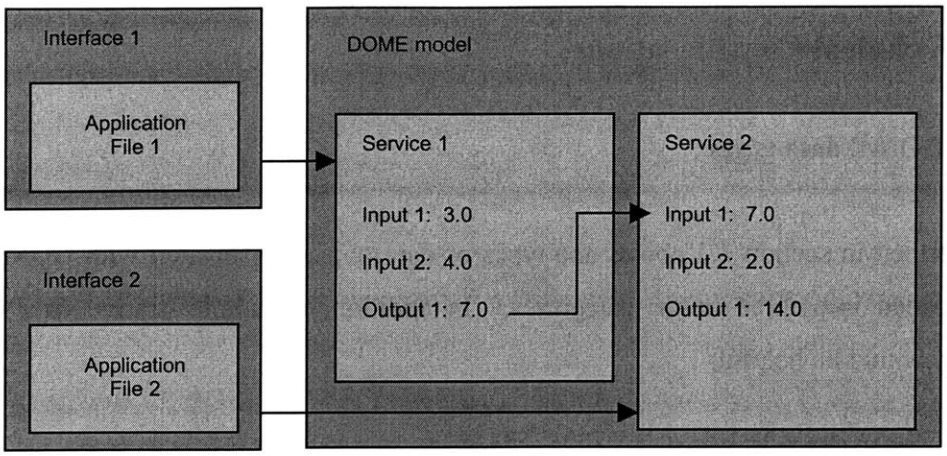

Figure 7: Simplified DOME data relationships

4.1.2 Database server

As the pioneer and industry leader in terms of market share, Oracle's relational database management system is often the default choice of database servers for corporate data management applications[Abbey, 1999]. Oracle provides a diverse range of tools and features that enhance the user's ability to manipulate and view data, including recent additions in Oracle 8i that address the arrival of the internet age [Greenwald, 1999]. However, as in many other areas of information technology, Microsoft has hit the market with a truly competitive product in SQL Server 2000.

Not only does SQL Server outperform Oracle in certain benchmarking studies [Transaction Processing Performance Council], but SQL Server carries with it the advantage of the Microsoft-style application interface (see Figure 8). The ease of use in terms of installing and using the SQL Server database management system is quite persuasive in deciding which database server to use, especially in a research situation where the proof of concept is more important than actual real world performance. That said, it appears that SQL Server is at least comparable to Oracle in terms of performance, and may even provide an edge in terms of tool sets; SQL server provides some innovative tools such as a graphical query analyzer that can help optimize SQL (Structured Query Language) statement processing.

4.1.3 Java and JDBC

The choices in terms of programming languages for implementing the DOME Data Manager basically come down to a choice between C++ or Java, both by virtue of their general pervasiveness in software development, as well as because of the DOME server and client's foundation in C++ and Java code. While C++ using the Microsoft Foundation Classes (MFC) could provide higher performance in terms of speed and user interface look and feel, Java was chosen for its ease of use in implementing database functionality, with a tolerable sacrifice in performance.

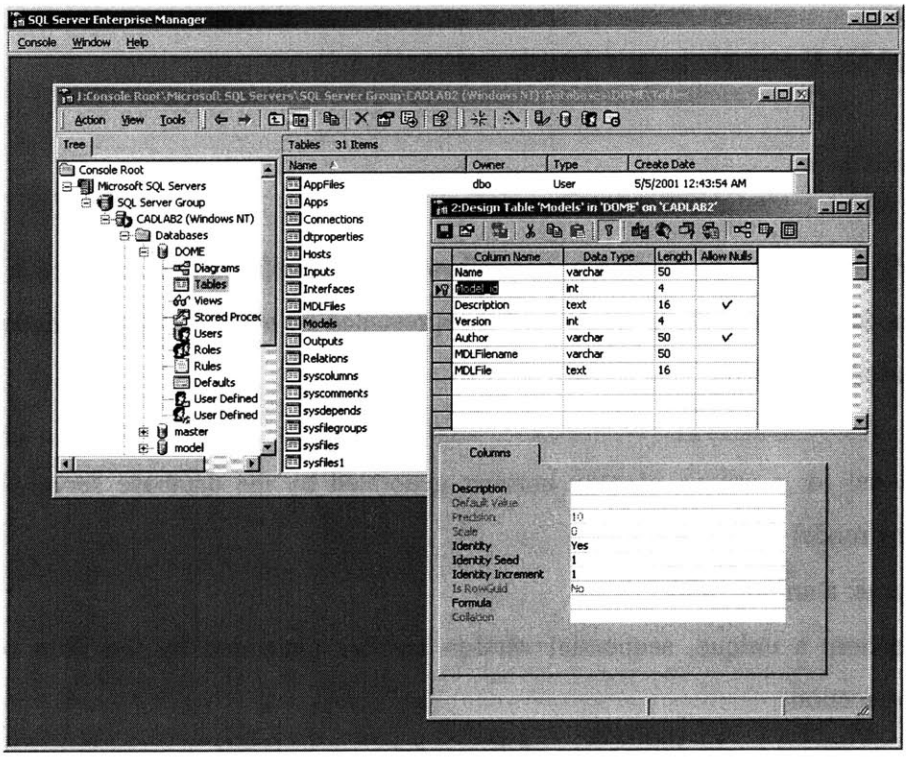

Figure 8: SQL Server's Windows-style interface

4.1.4 Integration of the Data Manager into DOME

Because Oculus Technologies holds rights to the core DOME implementation, access to the software guts of basic DOME functionality such as MDL file saving and model loading is restricted. While these difficulties may be reconcilable, the direction of this Data Manager implementation was to attempt to work within the boundaries of DOME source code availability. For example, access to the Java applet client is quite a bit simpler; so, rather than attempt to completely bypass existing MDL file export and loading functionality on the part of the server, this implementation simply modifies the way in which the client works with MDL files presented by the server.

4.2 DOME Data Manager Implementation

4.2.1 Metadata

The DOME Data Manager maintains the following set of information about each version of each model that is created. Each field is represented by a column in the Models table in the relational database:

- Model_id: a unique identity number generated by the database server for each new model
- Name: a unique model name
- Version: a unique, sequential version number generated by the Data Manager application
- Description: a text description of the model
- Author: the user who created the model
- MDLFilename: a unique filename generated by the Data Manager application for the MDL file from which the DOME server will load the model when requested
- MDLFile: the complete text of the MDL file

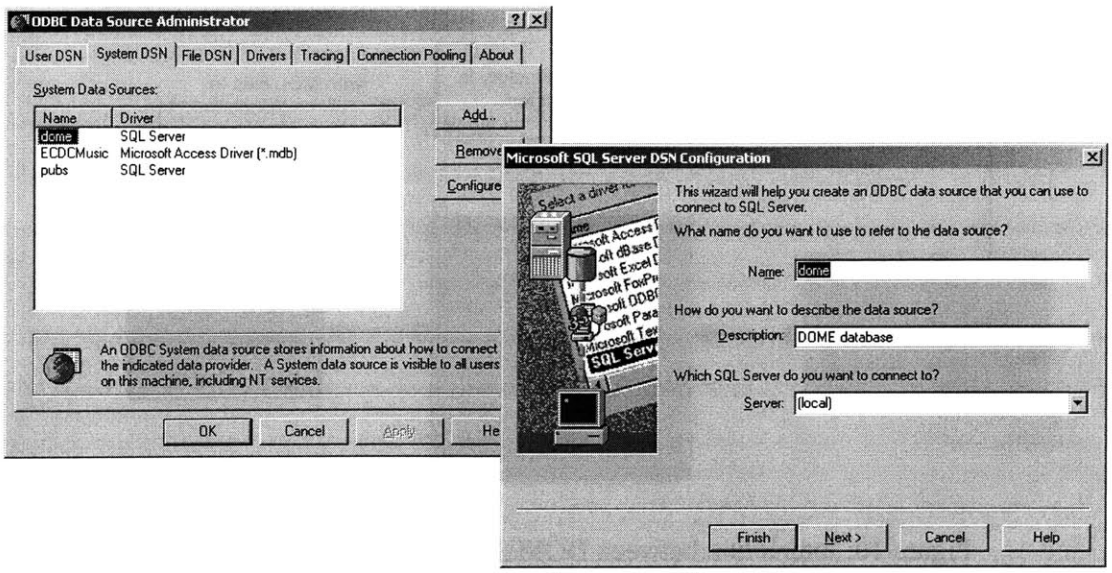

Figure 9: Microsoft Windows ODBC Data Source administration dialog

The MDLFile can be considered to be the physical representation of the model itself. By storing the MDL file in the database as the model, the Data Manager can work with the existing DOME server implementation, which loads all model data from MDL files.

4.2.2 Software code

JDBC (Java Database Connectivity) is comprised of a standard package in the Java 1.2.2 API. It provides classes that communicate with any database servers that support ODBC (Open Database Connectivity) standards. In Microsoft Windows 2000, ODBC databases are defined via the Data Sources control panel, available in the Administrative Tools program group in the Windows Start menu (see Figure 9). Once the ODBC data source has been defined, SQL queries and updates can be communicated from the Java application to the database via simple method calls.

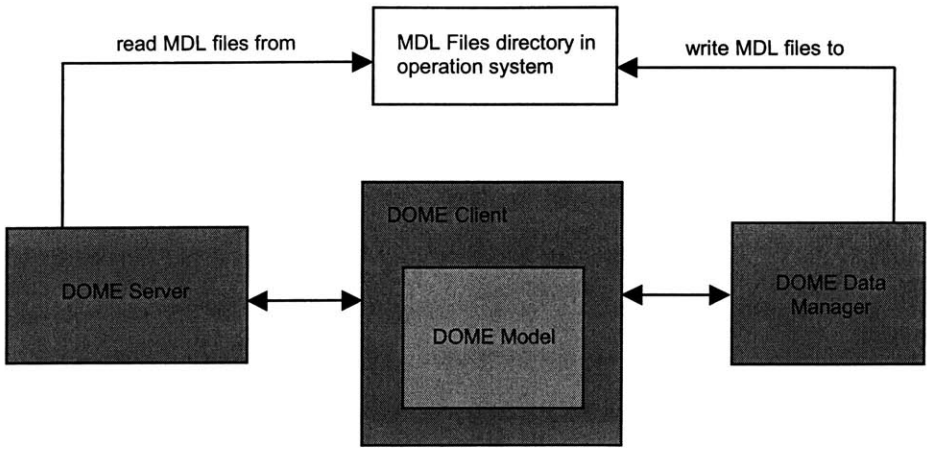

Figure 10: Interaction between DOME and DOME Data Manager

The main vehicle for database connectivity resides in a DataBase class. The graphical dialog in which model data is managed is defined by a ModelDialog class, which utilizes the DataBase class to pass custom statements to the database server. The Javadoc documentation for both classes are included in the Appendix.

4.2.3 Data Manager interactions with DOME and the user

Figure 10 illustrates how the DOME server, DOME client, and the Data Manager interact. Whenever a DOME model is created and saved in the database, an MDL file is automatically generated in a directory prescribed by the DOME server. When a DOME model is deleted from the database, the associated MDL file is also deleted. Thus, when the DOME server starts up and scans the directory for MDL files to load from, it is aware of all existing models.

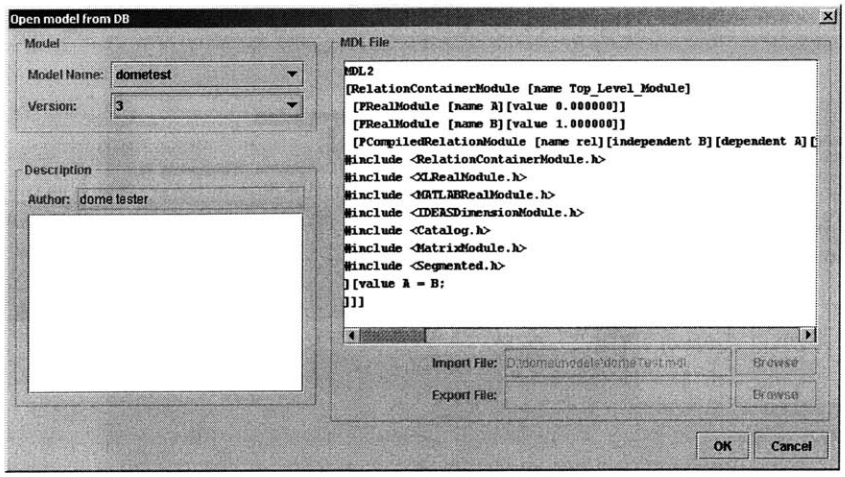

Figure 11: DOME Data Manager's Open Model dialog

The client works with stored models via the Data Manager dialog. The user can select any version of any model to open (see Figure 11).To open a model, the Data Manager writes the associated MDL file to the prescribed directory so that the DOME server can load the model properly. If the user then wishes to modify and save the model as a new version, the DataManager by default creates a new version of the model, although the user can choose to overwrite the last version of the model (see Figure 12). The Data Manager then creates an associated MDL file for the DOME server to load from if requested in the future. The user cannot overwite any version other than the last one because that would corrupt the model history by violating the version sequence. Similarly, the user can only delete all versions from a given version to the last version, inclusive (see Figure 13).

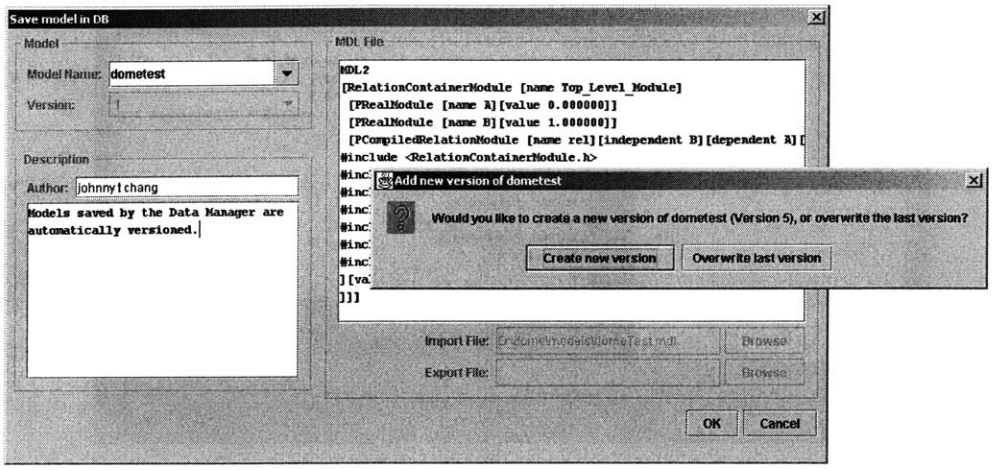

Figure 12: DOME Data Manager's Save Model dialog

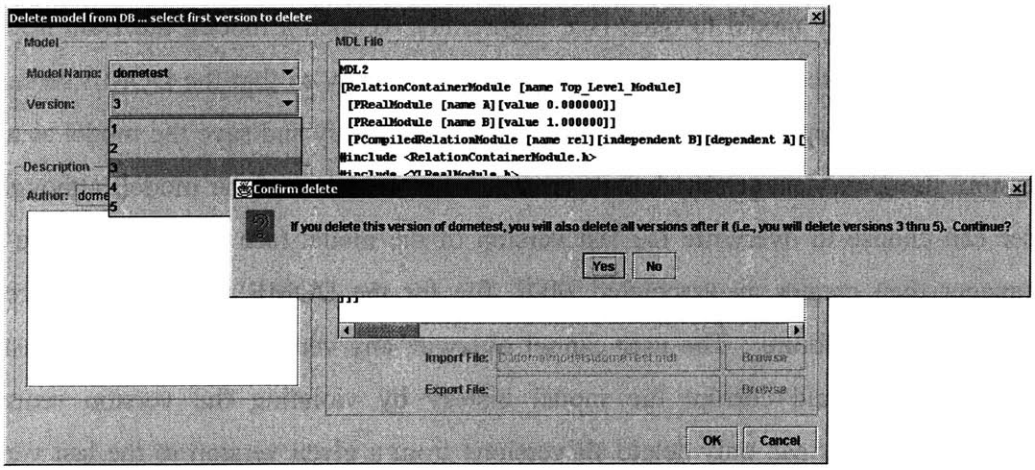

Figure 13: DOME Data Manager's Delete Model dialog

4.2.4 Benefits of data management provided by this implementation

The functionality provided by the Data Manager benefits DOME in several ways. It provides basic database connectivity sufficient for development of additional product data management features in the future. Rather than being a discrete tool as it is now, DOME can be more tightly integrated into the development process. Data flow within DOME can prompt events in the development process via product data management, much as data flow in conventional PDM systems can affect engineering process and procedures. In addition, this new data management functionality provides basic model metadata that can allow descriptive searches and model classification in the future. Lastly, on the simplest level, it provides the ability to warehouse models in an organized manner by allowing descriptive naming and a strict versioning system.

5 PROPOSED DATA MANAGER FOR IMPROVED PDM

With basic database connectivity implemented and tested via the ModelDialog versioning and model management class, more complex and beneficial database designs can be considered. While the lack of access to core DOME source code makes full integration of a data manager difficult, there is sufficient background in simply working with DOME models and simulations for considering a database schema that can provide beneficial product data management functionality to DOME. Essentially, such a schema would need to enable the storage of data, construction of metadata, and maintenance of relationships, that would allow the data manager to satisfy the goals listed at the top of section 4.

5.1 Proposed Schema

Based on the DOME data types described in section 4.1.1, the schema diagrammed in **Figure 14** is proposed. To start, services are stored in the database along with descriptive metadata. The DOME framework has been described as a marketplace for the exchange of services [Wallace, 2000]. By facilitating the discovery of and classification of these services with descriptive metadata, a DOME data manager encourages the growth of such a marketplace. In addition, proper service versioning within an organization can benefit the workflow by making service subscribers aware of the "freshness" of services that they may be using.

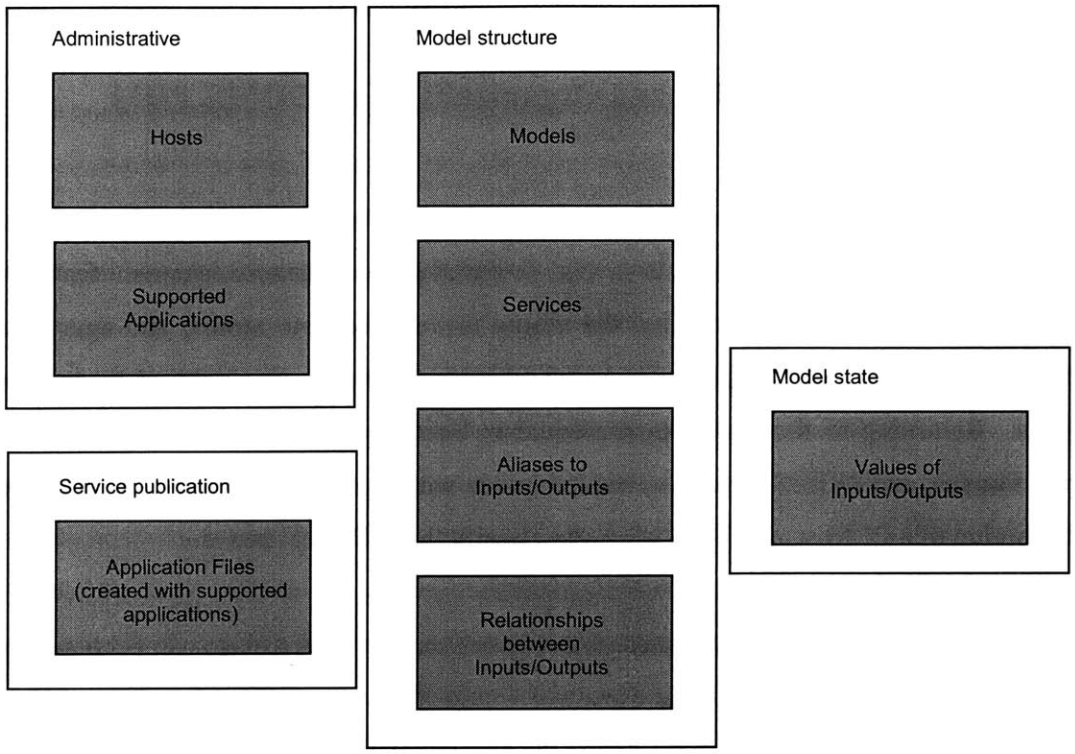

Figure 14: Basic components of proposed schema

In a larger sphere, each model is itself considered a data object that is versioned and described by metadata. At the same time, however, models are broken down into atomic components: containers, services, inputs/outputs, and relationships. This enables DOME to reconstruct saved models via the data manager. While this may appear to be overkill in light of the simplicity of storing model structures in simple text MDL files, it can provide benefits beyond what is easily achievable with a largely unformatted, plain text stream. Models can be categorized and described along additional axes; for example, being able to easily query for the services in a particular model may hint at the character of the model.

Lastly, the schema allows for the management of specific model states by saving specified variables in a model as versions of the model state. By keeping track of this data, the results of model simulations and analyses gain persistence, and those results can then be used to drive process and workflow.

6 FUTURE DEVELOPMENT AND BENEFITS OF PDM FOR DOME

6.1 Basic Data Description and Organization by a DOME PDM

The existence of an integrated, comprehensive data management system for DOME-related data objects opens the door to several potential product data management features. In terms of basic PDM functionality, the ability to describe both atomic and aggregate data objects within DOME allow for better classification and discovery of those data objects. Returning to the analogies presented in Section 2 that compare PDM data to labeled boxes, and DOME data to unlabeled boxes with working interfaces, the simplest contribution of a PDM would be to label and classify those DOME boxes.

In the complex space that a distributed object-based service marketplace can become, the lack of adequate data description and control would likely result in a chaotic jumble of black box data objects. To truly encourage the sharing of distributed expertise, within an organization or without, users must feel comfortable with their chances of finding useful, pertinent data when desired. This includes both individual services published by distributed sources, as well as complex models and model states that a person may decide to make available to others. Even regardless of the sharing aspects of the DOME framework, the ability to classify and organize DOME data components presents immediate benefits.

Furthermore, by breaking down DOME models into rows and relationships in a database, the inherent intelligence of a DOME model's service network is stored and available for querying. It can be envisioned that an engineer interested in motor torque outputs could query accessible databases for services that provide outputs with units in Newton-meters (Nm). More complex scenarios involving the evaluation of dependencies within a model by some type of intelligent agent in order to find desirable models or model states are not at all unlikely.

Ultimately, the data description functionality of the typical PDM system, when combined with the DOME framework, can result in a truly useful data marketplace, where the ability to view data classifications can help filter out the noise inherent in widely distributed data spaces. In addition, the ability to navigate data relationships would enable a certain degree of intelligence in evaluating the character of services and models.

6.2 Engineering Procedure Management by a DOME PDM

Beyond basic PDM functionality, a DOME data management system can allow DOME simulations to drive engineering procedures. After a user has defined design requirements and objective functions, a PDM that can track model states in terms of changes in designated input/output variables would be able to automatically version, archive, and potentially request release. In this way, the PDM can help track the complex changes in model states as a simulation is being run.

In a conventional PDM system, such triggers are generally administrative in nature. For example, an engineer signing off on a drawing that he is working on may trigger a review for release by a supervisor. Another example would be the release of a documentation set once all contributors have checked their work into the corporate vault. In the case of a PDM system within DOME, design releases can be more purely *data*-driven, where engineering procedures such as releases and reviews can be more tightly related to design specifications and requirements instead of business events.

6.3 Process and Workflow Management by a DOME PDM

While a conventional PDM can affect workflow by enabling the sharing of pre-release data among collaborators, the combination of PDM and DOME capabilities means that users can share not simply static data objects, but the visualization of design network simulations. The ability to archive, share, and version the effects of changes in complex

design interactions is invaluable in terms of decreasing the overhead of communicating design relationships in conventional settings. Such a change in the way that work can be done may indeed constitute the design paradigm mentioned in Section 1.1.

7 CONCLUSION

This thesis presented a product data management system for the Distributed Object-based Modeling Environment (DOME). Data utilization in a conventional product data management system as well as in DOME were described and compared. Specifically, the relational database was introduced as the foundation for PDM application layers in product development organizations. Such PDMs manage data by maintaining data attributes and data relationships that are relevant to a product life cycle, but do not allow the direct use of the data itself. DOME allows users to encapsulate data objects and share the inherent intelligence of those data objects in design simulation networks, but does not have PDM capabilities. A DOME data manager is proposed to provide DOME with the ability to maintain data attributes and relationships.

Microsoft's SQL Server 2000 was selected as the back-end for the DOME Data Manager. SQL Server has been proven in benchmarking tests to provide industry-leading performance. In addition, its standard Windows interface makes it an easier enterprise database application to learn and utilize. As an ODBC-compliant database server, SQL Server databases can be accessed via software through Java JDBC classes.

A DOME data manager was implemented in order to allow DOME to communicate with the SQL Server database. Presently, DOME is capable of storing its model structures in plain text files called MDL files. The files are maintained rather haphazardly, and do not allow easy searching because the only information that a user has about a model file is its filename. The implemented data manager allowed archiving, versioning, description, and retrieval of DOME models. In addition, the DOME data manager's basic database communication classes provided a basis for the future addition of PDM functionality to DOME.

A database schema that could provide DOME with PDM functionality was proposed and discussed. Such a schema would maintain attributes and relationships between several types of DOME data: service publication data, model structure data, model state data, and miscellaneous administrative data. At the simplest level, this would allow for persistence of both model structures and model states – that is, they can be recovered and modified as desired. However, the schema could also facilitate the sharing of services and models by enabling metadata searches.

The incorporation of product data management capabilities into a true data sharing environment such as DOME can truly affect the product development process. Expertise encapsulated in data objects can be shared in a distributed manner as DOME services. Available services can be discovered in an efficient way in a marketplace where available objects can be properly searched and organized. Models built with these services can also be shared and modified in an organized manner. Even the knowledge gained by manipulating these models can be maintained and shared.

By moving the management of data beyond mere data storage, PDM systems are capable of affecting both engineering procedures and engineering process. DOME goes further in enabling stored data by allowing the sharing of the expertise inherent in data objects through object encapsulation. By combining these two views of corporate product data and bringing stored data into an active marketplace, a PDM-enabled DOME can truly provide a paradigm in product development.

REFERENCES

Abbey, M, Corey, M J, Abramson, I (1999), *Oracle 8i: A Beginner's Guide*, Osborne McGraw-Hill, New York.

Ahmed, S (1991), "Transaction and Version Management in Object-oriented Database Management Systems for Collaborative Engineering Applications," M.S. Thesis, Department of Civil Engineering, Massachusetts Institute of Technology.

Blouin, G, "Introduction to Virtual Collaboration," PDMIC View & Markup Information Center, [http://www.pdmic.com/vmic/intro_vmic.shtml].

Bourke, R, "New PDM Apps Are More Capable at Managing Complex Data Relationships," Product Data Management Information Center, courtesy of CMstat Corporation, [http://www.pdmic.com/articles/artcmsta.html].

Conaway, J (1995), "Integrated Product Development: The Technology," Product Data Management Information Center, [http://www.pdmic.com/articles/artIPD1.html].

Finn, G A (1995), "Event-driven Knowledge-based Design," Ph.D. Thesis, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology.

Greenwald, R, Stackowiak, R, Stern, J (1999), *Oracle Essentials: Oracle8 and Oracle8i*, O'Reilly, Cambridge, Massachusetts.

Pahng, F, Senin, N, Wallace, D (1998), "Distributed Object-based Modeling and Evaluation of Design Problems," *Computer-aided Design*, vol. 30, no. 6, pp. 411-423.

Transaction Processing Performance Council, [http://www.tpc.org/].

Tsao, S S (1993), "An Overview of Product Information Management," Product Data Management Information Center, courtesy of Electronic Data Systems Corporation, [http://www.pdmic.com/articles/pacis.html].

Wallace, D, Abrahamson, S, Senin, N, Sferro, P (2000), "Integrated Design in a Service Marketplace," *Computer-aided Design*, vol. 32, no. 2, pp. 97-107.

Williams, C S (1994), "What Is Product Data Management and Why Should I Care?" *The Sun Observer*, November, 1994, pp. 25-27.

Williams, C S, "How Product Data Management Technology Has Evolved," Product Data Management Information Center, courtesy of Hewlett-Packard Company, [http://www.pdmic.com/evoltech.html].

APPENDIX

A.1 Class ModelDialog

data.ui.dialog

Class ModelDialog

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-java.awt.Window
                    |
                    +-java.awt.Dialog
                          |
                          +-javax.swing.JDialog
                                |
                                +-data.ui.dialog.ModelDialog
```

public class **ModelDialog**
extends javax.swing.JDialog
implements java.awt.event.ActionListener

The ModelDialog class provides an interface for working with models in the DOME database. The ModelDialog object should be provided with a DataBase object that provides basic database methods at instantiation. Users can save and open models to and from the database, as well as export and import models to and from text MDL files. The ModelDialog properly versions new models, and prevents haphazard deletion of versions from the middle of version series.

See Also:
Serialized Form

Inner classes inherited from class javax.swing.JDialog
`javax.swing.JDialog.AccessibleJDialog`

Fields inherited from class javax.swing.JDialog
`accessibleContext, rootPane, rootPaneCheckingEnabled`

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Constructor Summary
ModelDialog(javax.swing.JFrame owner, data.util.DataBase db) Instantiates a ModelDialog object with the given parent frame and DataBase.

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e)
boolean	**delete**() Deletes the model selected via the GUI dialog.
boolean	**exportMDLFile**() Exports a model to an MDL file.
boolean	**exportMDLFile**(java.io.File mdlFile) Exports a model to an MDL file.
int	**importMDLFile**() Imports a model from an MDL file.
int	**importMDLFile**(java.io.File mdlFile) Imports a model from an MDL file.
java.io.File	**openModel**() Creates a temporary MDL file for the selected model so that the model can be opened by DOME.
int	**saveModel**(java.io.File mdlFile) Save a model to the database with the proper version number.

Methods inherited from class javax.swing.JDialog
addImpl, createRootPane, dialogInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getJMenuBar, getLayeredPane, getRootPane, isRootPaneCheckingEnabled, paramString, processKeyEvent, processWindowEvent, remove, setContentPane, setDefaultCloseOperation, setGlassPane, setJMenuBar, setLayeredPane, setLayout, setLocationRelativeTo, setRootPane, setRootPaneCheckingEnabled, update

Methods inherited from class java.awt.Dialog
addNotify, dispose, getTitle, hide, isModal, isResizable, setModal, setResizable, setTitle, show

Methods inherited from class java.awt.Window
addWindowListener, applyResourceBundle, applyResourceBundle, finalize, getFocusOwner, getInputContext, getLocale, getOwnedWindows, getOwner, getToolkit, getWarningString, isShowing, pack, postEvent, processEvent, removeWindowListener, setCursor, toBack, toFront

Methods inherited from class java.awt.Container
add, add, add, add, add, addContainerListener, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getInsets, getLayout, getMaximumSize, getMinimumSize, getPreferredSize, insets, invalidate, isAncestorOf, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, removeNotify, setFont, validate, validateTree

Methods inherited from class java.awt.Component
action, add, addComponentListener, addFocusListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addPropertyChangeListener, addPropertyChangeListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, getBackground, getBounds, getBounds, getColorModel, getComponentOrientation, getCursor, getDropTarget, getFont, getFontMetrics, getForeground, getGraphics, getHeight, getInputMethodRequests, getLocation, getLocation, getLocationOnScreen, getName, getParent, getPeer, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isDisplayable, isDoubleBuffered, isEnabled, isFocusTraversable, isLightweight, isOpaque, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processInputMethodEvent, processMouseEvent, processMouseMotionEvent, remove, removeComponentListener, removeFocusListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, repaint, requestFocus, reshape, resize, resize, setBackground, setBounds, setBounds, setComponentOrientation, setDropTarget, setEnabled, setForeground, setLocale, setLocation, setLocation. setName. setSize. setSize. setVisible. show. size.

toString, transferFocus

Methods inherited from class java.lang.Object
clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

ModelDialog

```
public ModelDialog(javax.swing.JFrame owner,
                   data.util.DataBase db)
```

Instantiates a ModelDialog object with the given parent frame and DataBase.

Parameters:
`owner` - the parent JFrame that owns this dialog
`db` - the DataBase object through which this dialog communicates

Method Detail

importMDLFile

```
public int importMDLFile()
```

Imports a model from an MDL file. Filename and other parameters are collected via the GUI dialog.

Returns:
the database ID number of the new model

---

importMDLFile

```
public int importMDLFile(java.io.File mdlFile)
```

Imports a model from an MDL file. The given MDL file is loaded as the default one to import, but can be overriden via the GUI dialog.

Parameters:
`mdlFile` - the default MDL file from which to import
Returns:
the database ID number of the new model

---

openModel

`public java.io.File` **`openModel`**`()`

Creates a temporary MDL file for the selected model so that the model can be opened by DOME.

Returns:
the file object for the temporary MDL file

---

exportMDLFile

`public boolean` **`exportMDLFile`**`()`

Exports a model to an MDL file. Filename and other parameters are collected via the GUI dialog.

Returns:
true if successful, false otherwise

---

exportMDLFile

`public boolean` **`exportMDLFile`**`(java.io.File mdlFile)`

Exports a model to an MDL file. The given MDL file is used as the default one to export to, but this can be overriden via the GUI dialog.

Parameters:
`mdlFile` - the default MDL file to which to export
Returns:
true if successful, false otherwise

---

saveModel

`public int` **`saveModel`**`(java.io.File mdlFile)`

Save a model to the database with the proper version number.

Parameters:
`mdlFile` - the temporary MDL file from which to read the model
Returns:
the database ID of the saved model

---

delete

`public boolean` **`delete`**`()`

Deletes the model selected via the GUI dialog. Prevents deletion of versions in the middle of version series; i.e., only the latest version(s) of a model can be deleted.

Returns:
true if successful, false otherwise

---

actionPerformed

`public void` **`actionPerformed`**`(java.awt.event.ActionEvent e)`

Specified by:
actionPerformed in interface java.awt.event.ActionListener

A.2 Class DataBase

data.util

Class DataBase

```
java.lang.Object
  |
  +-data.util.DataBase
```

public class **DataBase**
extends java.lang.Object

The DataBase class handles all interactions with the database server. It establishes a connection with the DOME database and provides methods for executing SQL statements against that database.

Constructor Summary
DataBase(javax.swing.JFrame owner) Instantiates a DataBase object for the given owner frame, using the default JDBC bridge driver and the default dome database URL.
DataBase(javax.swing.JFrame owner, java.lang.String driverString, java.lang.String urlString) Instantiates a DataBase object for the given owner frame, using the given driver string (for a specific ODBC driver) and the given database URL (for a specific database).
DataBase(javax.swing.JFrame owner, java.lang.String driverString, java.lang.String urlString, java.lang.String user, java.lang.String pass) Instantiates a DataBase object for the given owner frame, using the given driver string (for a specific ODBC driver) and the given database URL (for a specific database).

Method Summary	
int	**addModel**(java.lang.String name, java.lang.String mdlFile, java.lang.String author, java.lang.String desc) Adds a row to the Models table, with the given model name, related temporary MDL filename (from which the DOME server reads model information), author, and description.

boolean	**deleteModel**(int id) Deletes the model with the given database ID.
boolean	**deleteModel**(java.lang.String name, int version) Deletes the given version of the model with the given name.
java.lang.String	**getMDLFilePath**() Returns the path of the directory in which temporary MDL files are placed so that the DOME server can read the model information.
java.lang.String	**getModelAuthor**(int id) Returns the author of the model with the given database ID.
java.lang.String	**getModelDesc**(int id) Returns the desription of the model with the given database ID.
int	**getModelID**(java.lang.String name, int version) Returns the database ID of the given version of the model with the given name.
java.lang.String	**getModelMDLFile**(int id) Returns the text of the MDL file that describes the model with the given database ID.
java.lang.String	**getModelMDLFilename**(int id) Returns the temporary MDL filename of the model with the given database ID.
java.lang.String	**getModelName**(int id) Returns the name of the model with the given database ID.
int	**getModelVersion**(int id) Returns the version of the model with the given database ID.
java.lang.Object[]	**getRowsArray**(java.lang.String table, java.lang.String column) Returns all rows of the given column in the given table, as an array of Objects.

`java.lang.Object[]`	**`getRowsArray`**`(java.lang.String table, java.lang.String column, java.lang.String expr)` Returns all rows of the given column in the given table that match the given expression, as an array of Objects.
`java.util.Vector`	**`getRowsVector`**`(java.lang.String table, java.lang.String column)` Returns all rows of the given column in the given table, as a Vector.
`java.util.Vector`	**`getRowsVector`**`(java.lang.String table, java.lang.String column, java.lang.String expr)` Returns all rows of the given column in the given table that match the given expression, as a Vector.
`boolean`	**`isModel`**`(java.lang.String name, int version)` Checks whether the given version of the model with the given name is in the database.
`boolean`	**`isName`**`(java.lang.String name)` Checks whether a model with the given name is in the database.
`java.sql.ResultSet`	**`sqlQuery`**`(java.lang.String sqlQuery)` Executes the given SQL query statement.
`boolean`	**`updateModelAuthor`**`(java.lang.String name, int version, java.lang.String newAuthor)` Changes the author associated with the given version of the model with the given name.
`boolean`	**`updateModelDesc`**`(java.lang.String name, int version, java.lang.String newDesc)` Changes the description associated with the given version of the model with the given name.
`boolean`	**`updateModelMDLFile`**`(java.lang.String name, int version, java.lang.String newMDLFile)` Changes the temporary MDL filename associated with the given version of the model with the given name.
`boolean`	**`updateModelName`**`(java.lang.String oldName, java.lang.String newName)` Changes the name associated with all versions of the model with the given name.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DataBase

```
public DataBase(javax.swing.JFrame owner)
```

Instantiates a DataBase object for the given owner frame, using the default JDBC bridge driver and the default dome database URL.

Parameters:

`owner` - the JFrame that will use this DataBase object

DataBase

```
public DataBase(javax.swing.JFrame owner,
                java.lang.String driverString,
                java.lang.String urlString)
```

Instantiates a DataBase object for the given owner frame, using the given driver string (for a specific ODBC driver) and the given database URL (for a specific database).

Parameters:

`owner` - the JFrame that will use this DataBase object
`driverString` - the driver-specific driver string
`urlString` - the driver- and database-specific database URL

DataBase

```
public DataBase(javax.swing.JFrame owner,
                java.lang.String driverString,
                java.lang.String urlString,
                java.lang.String user,
                java.lang.String pass)
```

Instantiates a DataBase object for the given owner frame, using the given driver string (for a specific ODBC driver) and the given database URL (for a specific database). Rather than showing a login dialog, automatically connects to the database with the given username and password.

Parameters:

`owner` - the JFrame that will use this DataBase object
`driverString` - the driver-specific driver string
`urlString` - the driver- and database-specific database URL

`user` - username
`pass` – password

Method Detail

sqlQuery

```
public java.sql.ResultSet sqlQuery(java.lang.String sqlQuery)
```

Executes the given SQL query statement. Does not execute DDL (data definition language) updates.

Parameters:
`sqlQuery` - the SQL string to be executed
Returns:
the ResultSet object containing the SQL query results

---

addModel

```
public int addModel(java.lang.String name,
                    java.lang.String mdlFile,
                    java.lang.String author,
                    java.lang.String desc)
```

Adds a row to the Models table, with the given model name, related temporary MDL filename (from which the DOME server reads model information), author, and description.

Parameters:
`name` - model name
`mdlFile` - MDL filename
`author` - author
`desc` - description
Returns:
database ID of the newly added model

---

deleteModel

```
public boolean deleteModel(int id)
```

Deletes the model with the given database ID.

Parameters:
`id` - database ID of the model to be deleted
Returns:
true if successful, false otherwise

---

deleteModel

```
public boolean deleteModel(java.lang.String name,
                           int version)
```

Deletes the given version of the model with the given name. Only allows deletion of the last version(s) of a given model, not a version within a series of versions.

Parameters:
`name` - name of the model to be deleted
`version` - version of model to be deleted
Returns:
true if successful, false otherwise

---

updateModelName

```
public boolean updateModelName(java.lang.String oldName,
                               java.lang.String newName)
```

Changes the name associated with all versions of the model with the given name.

Parameters:
`oldName` - the original name
`newName` - the new name to be used
Returns:
true if successful, false otherwise

---

updateModelAuthor

```
public boolean updateModelAuthor(java.lang.String name,
                                 int version,
                                 java.lang.String newAuthor)
```

Changes the author associated with the given version of the model with the given name.

Parameters:
`name` - model name
`version` - model version
`newAuthor` - new author
Returns:
true if successful, false otherwise

---

updateModelDesc

```
public boolean updateModelDesc(java.lang.String name,
                               int version,
                               java.lang.String newDesc)
```

Changes the description associated with the given version of the model with the given name.

Parameters:
`name` - model name
`version` - model version
`newDesc` - new description
Returns:
true if successful, false otherwise

---

updateModelMDLFile

```
public boolean updateModelMDLFile(java.lang.String name,
                                  int version,
                                  java.lang.String newMDLFile)
```

Changes the temporary MDL filename associated with the given version of the model with the given name.

Parameters:
`name` - model name
`version` - model version
`newMDLFile` - new MDL filename
Returns:
true if successful, false otherwise

---

isName

```
public boolean isName(java.lang.String name)
```

Checks whether a model with the given name is in the database.

Parameters:
`name` - model name
Returns:
true if the name is found in the database, false otherwise

---

isModel

```
public boolean isModel(java.lang.String name,
                       int version)
```

Checks whether the given version of the model with the given name is in the database.

Parameters:
`name` - model name
`version` - model version
Returns:
true if the model is found in the database, false otherwise

---

getMDLFilePath

```
public java.lang.String getMDLFilePath()
```

Returns the path of the directory in which temporary MDL files are placed so that the DOME server can read the model information.

Returns:
the path of the directory, as a String

---

getModelID

```
public int getModelID(java.lang.String name,
                      int version)
```

Returns the database ID of the given version of the model with the given name.

Parameters:
`name` - model name
`version` - model version
Returns:
database ID of the model

---

getModelName

```
public java.lang.String getModelName(int id)
```

Returns the name of the model with the given database ID.

Parameters:
`id` - database ID
Returns:
the name of the model with the given database ID

---

getModelVersion

```
public int getModelVersion(int id)
```

Returns the version of the model with the given database ID.

Parameters:
`id` - database ID
Returns:
version number of the model with the given database ID

---

getModelAuthor

`public java.lang.String` **`getModelAuthor`**`(int id)`

Returns the author of the model with the given database ID.

Parameters:
`id` - database ID
Returns:
the author, as a String

---

getModelDesc

`public java.lang.String` **`getModelDesc`**`(int id)`

Returns the desription of the model with the given database ID.

Parameters:
`id` - database ID
Returns:
the description, as a String

---

getModelMDLFilename

`public java.lang.String` **`getModelMDLFilename`**`(int id)`

Returns the temporary MDL filename of the model with the given database ID.

Parameters:
`id` - database ID
Returns:
the temporary MDL filename, as a String

---

getModelMDLFile

`public java.lang.String` **`getModelMDLFile`**`(int id)`

Returns the text of the MDL file that describes the model with the given database ID.

Parameters:
`if` - database ID
Returns:
the text of the MDL file, as a String

---

getRowsVector

```
public java.util.Vector getRowsVector(java.lang.String table,
                                      java.lang.String column)
```

Returns all rows of the given column in the given table, as a Vector.

Parameters:
`table` - table name
`column` - column name
Returns:
the Vector containing all requested rows

---

getRowsVector

```
public java.util.Vector getRowsVector(java.lang.String table,
                                      java.lang.String column,
                                      java.lang.String expr)
```

Returns all rows of the given column in the given table that match the given expression, as a Vector.

Parameters:
`table` - table name
`column` - column name
`expr` - the String boolean expression
Returns:
the Vector containing all requested rows

---

getRowsArray

```
public java.lang.Object[] getRowsArray(java.lang.String table,
                                       java.lang.String column)
```

Returns all rows of the given column in the given table, as an array of Objects.

Parameters:
`table` - table name
`column` - column name
Returns:
the array containing all requested rows as Objects

---

getRowsArray

```
public java.lang.Object[] getRowsArray(java.lang.String table,
                                       java.lang.String column,
                                       java.lang.String expr)
```

Returns all rows of the given column in the given table that match the given expression, as an array of Objects.

Parameters:
`table` - table name
`column` - column name
`expr` - the String boolean expression
Returns:
the array containing all requested rows as Objects